TITLE OF THE INVENTION

COMPILER, COMPILER APPARATUS AND COMPILATION METHOD

BACKGROUND OF THE INVENTION

(1) Field of the Invention

The present invention relates to a compiler that converts a source program described in a high-level language such as C++ language into a machine language, and especially to optimization by a compiler.

(2) Description of the Related Art

A high-performance compiler that can exploit effectively a high function that a processor has is much sought after as the function of the processor is greatly improved in recent year. In other words, the compiler that generates effectively advanced and specific instructions that the target processor executes is demanded.

For example, a processor that executes operation instructions in various fixed-point formats required for media processing such as digital signal processing and a high-performance processor that executes SIMD (Single Instruction Multiple Data) type instructions are developed. The compiler that targets such a processor is required to optimize a code size and execution speed by effectively generating the operation instructions in the various fixed-point formats and the SIMD type instructions.

It is not necessarily said, however, that a conventional compiler generates effectively advanced and specific instructions that the processor executes in regard to a source program described in a high-level language such as C++ language. Consequently, in the development of applications for media processing and others that require strict conditions in terms of the code size and the execution speed, a user has no choice but to describe critical points

in assembler instructions under the present conditions. But there is a problem that the programming by the assembler instructions not only requires more man-hours but also is vastly inferior in maintainability and portability compared with the development using a high-level language such as C++ language.

Additionally, the conventional compiler has within itself optimization processing for generating the advanced and specific instructions and the like that the processor executes. In other words, a processing module for optimization using effectively features of the target processor is included in the compiler itself and integrated. Consequently, when a function of the compiler is extended or the specifications of the target compiler are changed, it is necessary to reconfigure the whole compiler. There is a problem that an upgrading of the version and the like of the compiler must be repeated for each time.

SUMMARY OF THE INVENTION

In view of the foregoing, it is the first object of this invention to provide a compiler that can generate effectively advanced and specific instructions that a processor executes.

Moreover, it is the second object of this invention to provide a compiler that can make improvements through expanding functions and the like without repeating frequently upgrading of the version of the compiler itself.

The compiler according to the present invention translates a source program into a machine language program, the program including operation definition information in which operation that corresponds to a machine language instruction specific to a target processor is defined, the compiler comprising: a parser step of analyzing the source program; an intermediate code conversion step of converting the analyzed source program into intermediate codes; an optimization step of optimizing the converted

intermediate codes; and a code generation step of converting the optimized intermediate codes into machine language instructions, wherein the intermediate code conversion step includes: a detection sub-step of detecting whether or not any of the intermediate codes refer to the operation defined in the operation definition information; and a substitution sub-step of substituting the intermediate code with a corresponding machine language instruction, when the intermediate code is detected, and in the optimization step, the intermediate codes are optimized, the intermediate codes including the machine language instruction substituted for the intermediate code in the substitution sub-step.

For example, the program according to the present invention is made up of a header file included in the source program and the compiler that translates the source program into the machine language program; in the header file, a class made of data and methods is defined; t he compiler comprising: a parser step of analyzing the source program; an intermediate code conversion step of converting the analyzed source program into intermediate codes; an optimization step of optimizing the converted intermediate codes; and a code generation step of converting the optimized intermediate codes into machine language instructions, wherein the intermediate code conversion step includes: a detection sub-step of detecting whether or not any of the intermediate codes refer to the class defined in the header file; and a substitution sub-step of substituting the intermediate code with a corresponding machine language instruction, when the intermediate code is detected, and in the optimization step, the intermediate codes are optimized, the intermediate codes including the machine language instruction substituted for the intermediate code in the substitution sub-step.

Hereby, when there is a statement that refers to the class defined in the header file in the source program, the intermediate

code corresponding to the statement becomes a target of the optimization processing after the intermediate code is converted into a machine language instruction, and therefore, the intermediate code can be optimized together with machine language instructions in the vicinity. Additionally, since the compiler performs the optimization not only by the functional capability of the compiler itself (optimization processing) but also in connection with the definitions in the header file, the compiler can increase the statements that are the targets of the optimization and improves the level of the optimization.

Here, it is acceptable that the class defines a fixed point type, and in the detection sub-step, intermediate codes that use the fixed point type data are detected and it is also acceptable that the method in the class defines operators targeting the fixed point type data, in the detection sub-step, the detection is executed based on whether or not a set of the operator and the data type targeting an operation agrees with the definition in the method, and in the substitution step, an intermediate code whose set of the operator and the data type agrees with the definition is substituted with a corresponding machine language instruction.

Hereby, since the fixed point types and the operators defined by the header file are converted similarly with ordinary types into the corresponding intermediate codes and the machine language instructions by the compiler, by simply including the header file in the source program, similarly with the ordinary types, the user can declare and use the type corresponding to the fixed point mode specific to the target processor.

Moreover, the compiler according to the present invention comprises a header file included in the source program and the compiler that translates the source program into the machine language program; in the header file, a function is defined; the compiler comprising: a parser step of analyzing the source program;

an intermediate code conversion step of converting the analyzed source program into intermediate codes; an optimization step of optimizing the converted intermediate codes; and a code generation step of converting the optimized intermediate codes into machine language instructions, wherein the intermediate code conversion step includes: a detection sub-step of detecting whether or not any of the intermediate codes refer to the function defined in the header file; and a substitution sub-step of substituting the intermediate code with a corresponding machine language instruction, when the intermediate code is detected, and in the optimization step, the intermediate codes are optimized, the intermediate codes including the machine language instruction substituted for the intermediate code in the substitution sub-step.

Hereby, when there is a statement that refers to the function (the built-in function) defined in the header file in the source program, the intermediate code corresponding to the statement becomes a target of the optimization processing after the intermediate code is converted into a machine language instruction defined by the header file, and therefore, the intermediate code can be optimized together with machine language instructions in the vicinity. Furthermore, when the user wants to use high-functional instructions specific to the processor, he just need to describe that the header file is included in the source program and that a necessary built-in function is called in the source program. In other words, he is released from coding with assembler instructions.

As is described above, by the compiler according to the present invention, the high-functional and specific instructions that the target processor executes are effectively generated; the optimization is performed at a high level; and a flexible response by the header file such as function expansion become possible, and therefore, practical value of the compiler is extremely high especially as a development tool for a media processing application

for which strict specifications in both of the code size and the execution speed are required.

It should be noted that the present invention can be realized not only as the complier like this, but also as a compiler apparatus using the steps included in the program for the compiler as the steps and as a computer-readable recoding medium in which the characteristic compiler or the header file are recorded. Then, it goes without saying that the program and the data file like these can be widely distributed through a recording medium such as CD-ROM or a transmission medium such as Internet.

As further information about technical background to this application, Japanese patent application No. 2002-33668 filed August 2, 2002 is incorporated herein by reference.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other subjects, advantages and features of the invention will become apparent from the following description thereof taken in conjunction with the accompanying drawings that illustrate a specific embodiment of the invention. In the Drawings:

Fig.1 is a schematic block diagram showing a target processor of a compiler according to the present invention.

Fig.2 is a schematic diagram showing arithmetic and logic/comparison operation units of the processor.

Fig.3 is a block diagram showing a configuration of a barrel shifter of the processor.

Fig.4 is a block diagram showing a configuration of a converter of the processor.

Fig.5 is a block diagram showing a configuration of a divider of the processor.

Fig.6 is a block diagram showing a configuration of a multiplication/sum of products operation unit of the processor.

Fig.7 is a block diagram showing a configuration of an

instruction control unit of the processor.

Fig.8 is a diagram showing a configuration of general-purpose registers (R0~R31) of the processor.

Fig.9 is a diagram showing a configuration of a link register (LR) of the processor.

Fig.10 is a diagram showing a configuration of a branch register (TAR) of the processor.

Fig.11 is a diagram showing a configuration of a program status register (PSR) of the processor.

Fig.12 is a diagram showing a configuration of a condition flag register (CFR) of the processor.

Figs.13A and 13B are diagrams showing configurations of accumulators (M0, M1) of the processor.

Fig.14 is a diagram showing a configuration of a program counter (PC) of the processor.

Fig.15 is a diagram showing a configuration of a PC save register (IPC) of the processor.

Fig.16 is a diagram showing a configuration of a PSR save register (IPSR) of the processor.

Fig.17 is a timing diagram showing a pipeline behavior of the processor.

Fig.18 is a timing diagram showing each stage of the pipeline behavior of the processor at the time of executing an instruction.

Fig.19 is a diagram showing a parallel behavior of the processor.

Figs.20 is a diagram showing format of instructions executed by the processor.

Fig.21 is a diagram explaining an instruction belonging to a category "ALUadd (addition) system)".

Fig.22 is a diagram explaining an instruction belonging to a category "ALUsub (subtraction) system)".

Fig.23 is a diagram explaining an instruction belonging to a

category "ALUlogic (logical operation) system and the like".

Fig.24 is a diagram explaining an instruction belonging to a category "CMP (comparison operation) system".

Fig.25 is a diagram explaining an instruction belonging to a category "mul (multiplication) system".

Fig.26 is a diagram explaining an instruction belonging to a category "mac (sum of products operation) system".

Fig.27 is a diagram explaining an instruction belonging to a category "msu (difference of products) system".

Fig.28 is a diagram explaining an instruction belonging to a category "MEMld (load from memory) system".

Fig.29 is a diagram explaining an instruction belonging to a category "MEMstore (store in memory) system".

Fig.30 is a diagram explaining an instruction belonging to a category "BRA (branch) system".

Fig.31 is a diagram explaining an instruction belonging to a category "BSasl (arithmetic barrel shift) system and the like".

Fig.32 is a diagram explaining an instruction belonging to a category "BSlsr (logical barrel shift) system and the like".

Fig.33 is a diagram explaining an instruction belonging to a category "CNVvaln (arithmetic conversion) system".

Fig.34 is a diagram explaining an instruction belonging to a category "CNV (general conversion) system".

Fig.35 is a diagram explaining an instruction belonging to a category "SATvlpk (saturation processing) system".

Fig.36 is a diagram explaining an instruction belonging to a category "ETC (et cetera) system".

Fig. 37 is a function block diagram showing the configuration of a compiler according to the present invention.

Fig. 38 is a diagram showing a part of a list in the operator definition file.

Fig. 39 is a diagram showing a part of a list in the operator

definition file.

Fig. 40 is a diagram showing a part of a list in the operator definition file.

Fig. 41 is a diagram showing a part of a list in the operator definition file.

Fig. 42 is a diagram showing a part of a list in the operator definition file.

Fig. 43 is a diagram showing a part of a list in the operator definition file.

Fig. 44 is a diagram showing a part of a list in the operator definition file.

Fig. 45 is a diagram showing a part of a list in the operator definition file.

Fig. 46 is a diagram showing a part of a list in the operator definition file.

Fig. 47 is a diagram showing a part of a list in the operator definition file.

Fig. 48 is a diagram showing a part of a list in the operator definition file.

Fig. 49 is a diagram showing a part of a list in the operator definition file.

Fig. 50 is a diagram showing a part of a list in the operator definition file.

Fig. 51 is a diagram showing a part of a list in the operator definition file.

Fig. 52 is a diagram showing a part of a list in the operator definition file.

Fig. 53 is a diagram showing a part of a list in the operator definition file.

Fig. 54 is a diagram showing a part of a list in the operator definition file.

Fig. 55 is a diagram showing a part of a list in the operator

definition file.

Fig. 56 is a diagram showing a part of a list in the operator definition file.

Fig. 57 is a diagram showing a part of a list in the operator definition file.

Fig. 58 is a diagram showing a part of a list in the operator definition file.

Fig. 59 is a diagram showing a part of a list in the operator definition file.

Fig. 60 is a diagram showing a part of a list in the operator definition file.

Fig. 61 is a diagram showing a part of a list in the operator definition file.

Fig. 62 is a diagram showing a part of a list in the operator definition file.

Fig. 63 is a diagram showing a part of a list in the operator definition file.

Fig. 64 is a diagram showing a part of a list in the operator definition file.

Fig. 65 is a diagram showing a part of a list in the operator definition file.

Fig. 66 is a diagram showing a part of a list in the operator definition file.

Fig. 67 is a diagram showing a part of a list in the operator definition file.

Fig. 68 is a diagram showing a part of a list in the operator definition file.

Fig. 69 is a diagram showing a part of a list in the built-in function definition file.

Fig. 70 is a diagram showing a part of a list in the built-in function definition file.

Fig. 71 is a diagram showing a part of a list in the built-in

function definition file.

Fig. 72 is a diagram showing a part of a list in the built-in function definition file.

Fig. 73 is a flowchart showing the behaviors of the machine language instruction substitution unit.

Fig. 74 is a flowchart showing the behaviors of the argument optimization unit of the optimization unit.

Fig. 75 is a diagram showing an arithmetic tree to explain behaviors of the type conversion optimization unit of the optimization unit.

Fig. 76 is a diagram showing an example of a sample program to explain behaviors of the latency optimization unit.

Fig. 77 is a diagram explaining behaviors of the fixed point mode switch unit of the parser unit.

Fig. 78 is a diagram explaining a behavior verification technique using a class library.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

The compiler according to the present embodiment of the present invention is explained below in detail using the figures.

The compiler according to the present embodiment is a cross compiler that translates a source program described in a high-level language such as C/C++ languages into a machine language program that a specific processor (a target) can execute and has a characteristic that it can designate directives of optimization minutely concerning a code size and execution time of the machine language program to be generated.

(Processor)

For a start, an example of a target processor of the compiler according to the present embodiment is explained using Fig.1 through Fig. 36.

The processor that is the target of the compiler according to the present embodiment is, for example, a general-purpose processor that has been developed targeting on the field of AV media signal processing technology, and executable instructions has higher parallelism compared with ordinary microcomputers.

Fig.1 is a schematic block diagram showing the present processor. The processor 1 is an operational apparatus whose word length is 32 bits (a word) and is made up of an instruction control unit 10, a decoding unit 20, a register file 30, an operation unit 40, an I/F unit 50, an instruction memory unit 60, a data memory unit 70, an extended register unit 80, and an I/O interface unit 90. The operation unit 40 includes arithmetic and logic/comparison operation units 41~43, a multiplication/sum of products operation unit 44, a barrel shifter 45, a divider 46, and a converter 47 for performing SIMD instructions. The multiplication/sum of products operation unit 44 is capable of handling maximum of 65-bit accumulation so as not to decrease bit precision. The multiplication/sum of products operation unit 44 is also capable of executing SIMD instructions as in the case of the arithmetic and logic/comparison operation units 41~43. Furthermore, the processor 1 is capable of parallel execution of an arithmetic and logic/comparison operation instruction on maximum of three data elements.

Fig.2 is a schematic diagram showing the arithmetic and logic/comparison operation units 41~43. Each of the arithmetic and logic/comparison operation units 41~43 is made up of an ALU unit 41a, a saturation processing unit 41b, and a flag unit 41c. The ALU unit 41a includes an arithmetic operation unit, a logical operation unit, a comparator, and a TST. The bit widths of operation data to be supported are 8 bits (a bite. At this time, use four operation units in parallel), 16 bits (a half word. At this time, use two operation units in parallel) and 32 bits (a word. At this

time, process 32-bit data using all operation units). For a result of an arithmetic operation, the flag unit 41c and the like detect an overflow and generate a condition flag. For a result of each of the operation units, the comparator and the TST, an arithmetic shift right, saturation by the saturation processing unit 41b, the detection of maximum/minimum values, absolute value generation processing are performed.

Fig.3 is a block diagram showing the configuration of the barrel shifter 45. The barrel shifter 45, which is made up of selectors 45a and 45b, a higher bit shifter 45c, a lower bit shifter 45d, and a saturation processing unit 45e, executes an arithmetic shift of data (shift in the 2's complement number system) or a logical shift of data (unsigned shift). Usually, 32-bit or 64-bit data are inputted to and outputted from the barrel shifter 45. The amount of shift of target data stored in the registers 30a and 30b are specified by another register or according to its immediate value. An arithmetic or logical shift in the range of left 63 bits and right 63 bits is performed for the data, which is then outputted in an input bit length.

The barrel shifter 45 is capable of shifting 8-, 16-, 32-, and 64-bit data in response to a SIMD instruction. For example, the barrel shifter 45 can shift four pieces of 8-bit data in parallel.

Arithmetic shift, which is a shift in the 2's complement number system, is performed for aligning decimal points at the time of addition and subtraction, for multiplying a power of 2 (2, the 2^{nd} power of 2, the -1^{st} power of 2) and other purposes.

Fig.4 is a block diagram showing the configuration of the converter 47. The converter 47 is made up of a saturation block (SAT) 47a, a BSEQ block 47b, an MSKGEN block 47c, a VSUMB block 47d, a BCNT block 47e, and an IL block 47f.

The saturation block (SAT) 47a performs saturation processing for input data. Having two blocks for the saturation

processing of 32-bit data makes it possible to support a SIMD instruction executed for two data elements in parallel.

The BSEQ block 47b counts consecutive 0s or 1s from the MSB.

The MSKGEN block 47c outputs a specified bit segment as 1, while outputting the others as 0.

The VSUMB block 47d divides the input data into specified bit widths, and outputs their total sum.

The BCNT block 47e counts the number of bits in the input data specified as 1.

The IL block 47f divides the input data into specified bit widths, and outputs a value resulted from exchanging the position of each data block.

Fig.5 is a block diagram showing the configuration of the divider 46. Letting a dividend be 64 bits and a divisor be 32 bits, the divider 46 outputs 32 bits as a quotient and a modulo, respectively. 34 cycles are involved for obtaining a quotient and a modulo. The divider 46 can handle both singed and unsigned data. It should be noted, however, that an identical setting is made concerning the presence/absence of signs of data serving as a dividend and a divisor. Also, the divider 46 has the capability of outputting an overflow flag, and a 0 division flag.

Fig.6 is a block diagram showing the configuration of the multiplication/sum of products operation unit 44. The multiplication/sum of products operation unit 44, which is made up of two 32-bit multipliers (MUL) 44a and 44b, three 64-bit adders (Adder) 44c~44e, a selector 44f and a saturation processing unit (Saturation) 44g, performs the following multiplications and sums of products:

32x32-bit signed multiplication, sum of products, and difference of products;

・32x32-bit unsigned multiplication;

· 16x16-bit signed multiplication, sum of products, and difference of products performed on two data elements in parallel; and

· 32x16-bit t signed multiplication, sum of products, and difference of products performed on two data elements in parallel;

The above operations are performed on data in integer and fixed point format (h1, h2, w1, and w2). Also, the results of these operations are rounded and saturated.

Fig.7 is a block diagram showing the configuration of the instruction control unit 10. The instruction control unit 10, which is made up of an instruction cache 10a, an address management unit 10b, instruction buffers 10c~10e, a jump buffer 10f, and a rotation unit (rotation) 10g, issues instructions at ordinary times and at branch points. Having three 128-bit instruction buffers (the instruction buffers 10c~10e) makes it possible to support the maximum number of parallel instruction execution. Regarding branch processing, the instruction control unit 10 stores in advance a branch destination address in the below-described TAR register via the jump buffer 10f and others before performing a branch (settar instruction). The branch is performed using the branch destination address stored in the TAR register.

It should be noted that the processor 1 is a processor employing the VLIW architecture. The VLIW architecture is an architecture allowing a plurality of instructions (e.g. load, store, operation, and branch) to be stored in a single instruction word, and such instructions to be executed all at once. By programmers describing a set of instructions which can be executed in parallel as a single issue group, it is possible for such issue group to be processed in parallel. In this specification, the delimiter of an issue group is indicated by ";;". Notational examples are described below.

(Example 1)

```
mov r1, 0x23;;
```

This instruction description indicates that only an instruction "mov" shall be executed.

(Example 2)

```
mov r1, 0x38
add r0, r1, r2
sub r3, r1, r2;;
```

These instruction descriptions indicate that three instructions of "mov", "add" and "sub" shall be executed in parallel.

The instruction control unit 10 identifies an issue group and sends it to the decoding unit 20. The decoding unit 20 decodes the instructions in the issue group, and controls resources required for executing such instructions.

Next, an explanation is given for registers included in the processor 1.

Table 1 below lists a set of registers of the processor 1.

[Table 1]

Register name	Bit width	No. of registers	Usage
R0~R31	32 bits	32	General-purpose registers. Used as data memory pointer, data storage and the like when operation instruction is executed.
TAR	32 bits	1	Branch register. Used as branch address storage at branch point.
LR	32 bits	1	Link register.
SVR	16 bits	2	Save register. Used for saving condition flag (CFR) and various modes.
M0~M1 (MH0:ML0~ MH1~ML1)	64 bits	2	Operation registers. Used as data storage when operation instruction is executed.

Table 2 below lists a set of flags (flags managed in a

condition flag register and the like described later) of the processor 1.

[Table 2]

Flag name	Bit width	No. of flags	Usage
C0~C7	1	8	Condition flags. Indicate if condition is established or not.
VC0~VC3	1	4	Condition flags for media processing extension instruction. Indicate if condition is established or not.
OVS	1	1	Overflow flag. Detects overflow at the time of operation.
CAS	1	1	Carry flag. Detects carry at the time of operation.
BPO	5	1	Specifies bit position. Specifies bit positions to be processed when mask processing instruction is executed.
ALN	2	1	Specified byte alignment.
FXP	1	1	Fixed point operation mode.
UDR	32	1	Undefined register.

Fig.8 is a diagram showing the configuration of the general-purpose registers (R0~R31) 30a. The general-purpose registers (R0~R31) 30a are a group of 32-bit registers that constitute an integral part of the context of a task to be executed and that store data or addresses. It should be noted that the general-purpose registers R30 and R31 are used by hardware as a global pointer and a stack pointer respectively.

Fig.9 is a diagram showing the configuration of a link register (LR) 30c. In connection with this link register (LR) 30c, the processor 1 also has a save register (SVR) not illustrated in the diagram. The link register (LR) 30c is a 32-bit register for storing a return address at the time of a function call. It should be noted that

the save register (SVR) is a 16-bit register for saving a condition flag (CFR.CF) of the condition flag register at the time of a function call. The link register (LR) 30c is used also for the purpose of increasing the speed of loops, as in the case of a branch register (TAR) to be explained later. 0 is always read out as the lower 1 bit, but 0 must be written at the time of writing.

For example, when "call" (brl, jmpl) instruction is executed, the processor 1 saves a return address in the link register (LR) 30c and saves a condition flag (CFR.CF) in the save register (SVR). When "jmp" instruction is executed, the processor 1 fetches the return address (branch destination address) from the link register (LR) 30c, and returns a program counter (PC). Furthermore, when "ret (jmpr)" instruction is executed, the processor 1 fetches the branch destination address (return address) from the link register (LR) 30c, and stores (restores) it in/to the program counter (PC). Moreover, the processor 1 fetches the condition flag from the save register (SVR) so as to store (restores) it in/to a condition flag area CFR.CF in the condition flag register (CFR) 32.

Fig.10 is a diagram showing the configuration of the branch register (TAR) 30d. The branch register (TAR) 30d is a 32-bit register for storing a branch target address, and used mainly for the purpose of increasing the speed of loops. 0 is always read out as the lower 1 bit, but 0 must be written at the time of writing.

For example, when "jmp" and "jloop" instructions are executed, the processor 1 fetches a branch destination address from the branch register (TAR) 30d, and stores it in the program counter (PC). When the instruction indicated by the address stored in the branch register (TAR) 30d is stored in a branch instruction buffer, a branch penalty will be 0. An increased loop speed can be achieved by storing the top address of a loop in the branch register (TAR) 30d.

Fig.11 is a diagram showing the configuration of a program status register (PSR) 31. The program status register (PSR) 31,

which constitutes an integral part of the context of a task to be executed, is a 32-bit register for storing the following processor status information:

Bit SWE: indicates whether the switching of VMP (Virtual Multi-Processor) to LP (Logical Processor) is enabled or disabled. "0" indicates that switching to LP is disabled and "1" indicates that switching to LP is enabled.

Bit FXP: indicates a fixed point mode. "0" indicates the mode 0 (the mode in which an arithmetic operation is performed supposing that there is the decimal point between the MSB and the first bit from the MSB. Hereafter, also called "_1 system") and "1" indicates the mode 1 (the mode in which an arithmetic operation is performed supposing that there is the decimal point between the first bit from the MSB and the second bit from the MSB. Hereafter, also called "_2 system").

Bit IH: is an interrupt processing flag indicating that maskable interrupt processing is ongoing or not. "1" indicates that there is an ongoing interrupt processing and "0" indicates that there is no ongoing interrupt processing. This flag is automatically set on the occurrence of an interrupt. This flag is used to make a distinction of whether interrupt processing or program processing is taking place at a point in the program to which the processor returns in response to "rti" instruction.

Bit EH: is a flag indicating that an error or an NMI is being processed or not. "0" indicates that error/NMI interrupt processing is not ongoing and "1" indicates that error/NMI interrupt processing is ongoing. This flag is masked if an asynchronous error or an NMI occurs when EH=1. Meanwhile, when VMP is enabled, plate switching of VMP is masked.

Bit PL [1:0]: indicates a privilege level. "00" indicates the privilege level 0, i.e. the processor abstraction level, "01" indicates the privilege level 1 (non-settable), "10" indicates the privilege level

2, i.e. the system program level, and "11" indicates the privilege level 3, i.e. the user program level.

Bit LPIE3: indicates whether LP-specific interrupt 3 is enabled or disabled. "1" indicates that an interrupt is enabled and "0" indicates that an interrupt is disabled.

Bit LPIE2: indicates whether LP-specific interrupt 2 is enabled or disabled. "1" indicates that an interrupt is enabled and "0" indicates that an interrupt is disabled.

Bit LPIE1: indicates whether LP-specific interrupt 1 is enabled or disabled. "1" indicates that an interrupt is enabled and "0" indicates that an interrupt is disabled.

Bit LPIE0: indicates whether LP-specific interrupt 0 is enabled or disabled. "1" indicates that an interrupt is enabled and "0" indicates that an interrupt is disabled.

Bit AEE: indicates whether a misalignment exception is enabled or disabled. "1" indicates that a misalignment exception is enabled and "0" indicates that a misalignment exception is disabled.

Bit IE: indicates whether a level interrupt is enabled or disabled. "1" indicates that a level interrupt is enabled and "0" indicates a level interrupt is disabled.

Bit IM [7:0]: indicates an interrupt mask, and ranges from levels 0~7, each being able to be masked at its own level. Level 0 is the highest level. Of interrupt requests which are not masked by any IMs, only the interrupt request with the highest level is accepted by the processor 1. When an interrupt request is accepted, levels below the accepted level are automatically masked by hardware. IM[0] denotes a mask of level 0, IM[1] a mask of level 1, IM[2] a mask of level 2, IM[3] a mask of level 3, IM[4] a mask of level 4, IM[5] a mask of level 5, IM[6] a mask of level 6, and IM[7] a mask of level 7.

reserved: indicates a reserved bit. 0 is always read out. 0 must be written at the time of writing.

Fig.12 is a diagram showing the configuration of the condition flag register (CFR) 32. The condition flag register (CFR) 32, which constitutes an integral part of the context of a task to be executed, is a 32-bit register made up of condition flags, operation flags, vector condition flags, an operation instruction bit position specification field, and a SIMD data alignment information field.

Bit ALN [1:0]: indicates an alignment mode. An alignment mode of "valnvc" instruction is set.

Bit BPO [4:0]: indicates a bit position. It is used in an instruction that requires a bit position specification.

Bit VC0~VC3: are vector condition flags. Starting from a byte on the LSB side or a half word through to the MSB side, each corresponds to a flag ranging from VC0 through to VC3.

Bit OVS: is an overflow flag (summary). It is set on the detection of saturation and overflow. If not detected, a value before the instruction is executed is retained. Clearing of this flag needs to be carried out by software.

Bit CAS: is a carry flag (summary). It is set when a carry occurs under "addc" instruction, or when a borrow occurs under "subc" instruction. If there is no occurrence of a carry under "addc" instruction, or a borrow under "subc" instruction, a value before the instruction is executed is retained. Clearing of this flag needs to be carried out by software.

Bit C0~C7: are condition flags, which indicate a condition (TRUE/FALSE) in an execution instruction with a condition. The correspondence between the condition of the execution instruction with the condition and the bits C0~C7 is decided by the predicate bit included in the instruction. It should be noted that the value of the flag C7 is always 1. A reflection of a FALSE condition (writing of 0) made to the flag C7 is ignored.

reserved: indicates a reserved bit. 0 is always read out. 0 must be written at the time of writing.

Figs.13A and 13B are diagrams showing the configurations of accumulators (M0, M1) 30b. Such accumulators (M0, M1) 30b, which constitute an integral part of the context of a task to be executed, are made up of a 32-bit register MHO-MH1 (register for multiply and divide/sum of products (the higher 32 bits)) shown in Fig.13A and a 32-bit register MLO-ML1 (register for multiply and divide/sum of products (the lower 32 bits)) shown in Fig.13B.

The register MHO-MH1 is used for storing the higher 32 bits of operation results at the time of a multiply instruction, while used as the higher 32 bits of the accumulators at the time of a sum of products instruction. Moreover, the register MHO-MH1 can be used in combination with the general-purpose registers in the case where a bit stream is handled. Meanwhile, the register MLO-ML1 is used for storing the lower 32 bits of operation results at the time of a multiply instruction, while used as the lower 32 bits of the accumulators at the time of a sum of products instruction.

Fig.14 is a diagram showing the configuration of a program counter (PC) 33. This program counter (PC) 33, which constitutes an integral part of the context of a task to be executed, is a 32-bit counter that holds the address of an instruction being executed.

Fig.15 is a diagram showing the configuration of a PC save register (IPC) 34. This PC save register (IPC) 34, which constitutes an integral part of the context of a task to be executed is a 32-bit register.

Fig.16 is a diagram showing the configuration of a PSR save register (IPSR) 35. This PSR save register (IPSR) 35, which constitutes an integral part of the context of a task to be executed, is a 32-bit register for saving the program status register (PSR) 31. 0 is always read out as a part corresponding to a reserved bit, but 0 must be written at the time of writing.

Next, an explanation is given for the memory space of the processor 1, which is the target of the compiler according to the

present embodiment. For example, in the processor 1, a linear memory space with a capacity of 4 GB is divided into 32 segments, and an instruction SRAM (Static RAM) and a data SRAM are allocated to 128-MB segments. With a 128-MB segment serving as one block, an object block to be accessed is set in a SAR (SRAM Area Register). A direct access is made to the instruction SRAM/data SRAM when the accessed address is a segment set in the SAR, but an access request shall be issued to a bus controller (BCU) when such address is not a segment set in the SAR. An on chip memory (OCM), an external memory, an external device, an I/O port and the like are connected to the BUC. Data reading/writing from and to these devices is possible.

Fig.17 is a timing diagram showing the pipeline behavior of the processor 1, which is the target of the compiler according to the present embodiment. As illustrated in the diagram, the pipeline of the processor 1 basically consists of the following five stages: instruction fetch; instruction assignment (dispatch); decode; execution; and writing.

Fig.18 is a timing diagram showing each stage of the pipeline behavior of the processor 1 at the time of executing an instruction. In the instruction fetch stage, an access is made to an instruction memory which is indicated by an address specified by the program counter (PC) 33, and an instruction is transferred to the instruction buffers 10c~10e and the like. In the instruction assignment stage, the output of branch destination address information in response to a branch instruction, the output of an input register control signal, the assignment of a variable length instruction are carried out, which is followed by the transfer of the instruction to an instruction register (IR). In the decode stage, the IR is inputted to the decoding unit 20, and an operation unit control signal and a memory access signal are outputted. In the execution stage, an operation is executed and the result of the operation is outputted either to the

data memory or the general-purpose registers (R0~R31) 30a. In the writing stage, a value obtained as a result of data transfer, and the operation results are stored in the general-purpose registers.

The VLIW architecture of the processor 1, which is the target of the compiler according to the present embodiment, allows parallel execution of the above processing on maximum of three data elements. Therefore, the processor 1 performs the behavior shown in Fig.18 in parallel at the timing shown in Fig.19.

Next, an explanation is given for a set of instructions executed by the processor 1 with the above configuration.

Tables 3~5 list categorized instructions to be executed by the processor 1, which is the target of the compiler according to the present embodiment.

[Table 3]

Category	Operation unit	Instruction operation code
Memory transfer instruction (load)	M	ld,ldh,ldhu,ldb,ldbu,ldp,ldhp,ldbp,ldbh,ldbuh,ldbhp,ldbuhp
Memory transfer instruction (store)	M	st,sth,stb,stp,sthp,stbp,stbh,stbhp
Memory transfer instruction (others)	M	dpref,ldstb
External register transfer instruction	M	rd,rde,wt,wte
Branch instruction	B	br,brl,call,jmp,jmpl,jmpr,ret,jmpf,jloop,setbb,setlr,settar
Software interrupt instruction	B	rti,pi0,pi0l,pi1,pi1l,pi2,pi2l,pi3,pi3l,pi4,pi4l,pi5,pi5l,pi6,pi6l,pi7,pi7l,sc0,sc1,sc2,sc3,sc4,sc5,sc6,sc7
VMP/interrupt control instruction	B	intd,inte,vmpsleep,vmpsus,vmpswd,vmpswe,vmpwait

Category	Operation unit	Instruction operation code
Arithmetic operation instruction	A	abs,absvh,absvw,add,addarvw,addc,addmsk,adds,addsr,addu,addvh,addvw,neg,negvh,negvw,rsub,s1add,s2add,sub,subc,submsk,subs,subvh,subvw;max,min
Logical operation instruction	A	and,andn,or,sethi,xor,not
Compare instruction	A	cmpCC,cmpCCa,cmpCCn,cmpCCo,tstn,tstna,tstnn,tstno,tstz,tstza,tstzn,tstzo
Move instruction	A	mov,movcf,mvclcas,mvclovs,setlo,vcchk
NOP instruction	A	nop
Shift instruction1	S1	asl,aslvh,aslvw,asr,asrvh,asrvw,lsl,lsr,rol,ror
Shift instruction2	S2	aslp,aslpvw,asrp,asrpvw,lslp,lsrp

[Table 4]

Category	Operation unit	Instruction operation code
Extraction instruction	S2	ext,extb,extbu,exth,exthu,extr,extru,extu
Mask instruction	C	msk,mskgen
Saturation instruction	C	sat12,sat9,satb,satbu,sath,satw
Conversion instruction	C	valn,valn1,valn2,valn3,valnvc1,valnvc2,valnvc3,valnvc4,vhpkb,vhpkh,vhunpkb,vhunpkh,vintlhb,vintlhh,vintllb,vintllh,vlpkb,vlpkbu,vlpkh,vlpkhu,vlunpkb,vlunpkbu,vlunpkh,vlunpkhu,vstovb,vstovh,vunpk1,vunpk2,vxchngh,vexth
Bit count instruction	C	bcnt1,bseq,bseq0,bseq1

Category	Operation unit	Instruction operation code
Others	C	byterev,extw,mskbrvb,mskbrvh,rndvh,movp
Multiply instruction1	X1	fmulhh,fmulhhr,fmulhw,fmulhww,hmul,lmul
Multiply instruction2	X2	fmulww,mul,mulu
Sum of products instruction1	X1	fmachh,fmachhr,fmachw,fmachww,hmac,lmac
Sum of products instruction2	X2	fmacww,mac
Difference of products instruction1	X1	fmsuhh,fmsuhhr,fmsuhw,fmsuww,hmsu,lmsu
Difference of products instruction2	X2	fmsuww,msu
Divide instruction	DIV	div,divu
Debugger instruction	DBGM	dbgm0,dbgm1,dbgm2,dbgm3

5

[Table 5]

Category	Operation unit	Instruction operation code
SIMD arithmetic operation instruction	A	vabshvh,vaddb,vaddh,vaddhvc,vaddhvh,vaddrhvc,vaddsb,vaddsh,vaddsrb,vaddsrh,vasubb,vcchk,vhaddh,vhaddhvh,vhsubh,vhsubhvh,vladdh,vladdhvh,vlsubh,vlsubhvh,vnegb,vnegh,vneghvh,vsaddb,vsaddh,vsgnh,vsrsubb,vsrsubh,vssubb,vssubh,vsubb,vsubh,vsubhvh,vsubsh,vsumh,vsumh2,vsumrh2,vxaddh,vxaddhvh,vxsubh,vxsubhvh,vmaxb,vmaxh,vminb,vminh,vmovt,vse

Category	Operation unit	Instruction operation code
		I
SIMD compare instruction	A	vcmpeqb,vcmpeqh,vcmpgeb,vcmpgeh,vcmpgtb,vcmpgth,vcmpleb,vcmpleh,vcmpltb,vcmplth,vcmpneb,vcmpneh,vscmpeqb,vscmpeqh,vscmpgeb,vscmpgeh,vscmpgtb,vscmpgth,vscmpleb,vscmpleh,vscmpltb,vscmplth,vscmpneb,vscmpneh
SIMD shift instruction1	S1	vaslb,vaslh,vaslvh,vasrb,vasrh,vasrvh,vlslb,vlslh,vlsrb,vlsrh,vrolb,vrolh,vrorb,vrorh
SIMD shift instruction2	S2	vasl,vaslvw,vasr,vasrvw,vlsl,vlsr
SIMD saturation instruction	C	vsath,vsath12,vsath8,vsath8u,vsath9
Other SIMD instruction	C	vabssumb,vrndvh
SIMD multiply instruction	X2	vfmulh,vfmulhr,vfmulw,vhfmulh,vhfmulhr,vhfmulw,vhmul,vlfmulh,vlfmulhr,vlfmulw,vlmul,vmul,vpfmulhww,vxfmulh,vxfmulhr,vxfmulw,vxmul
SIMD sum of products instruction	X2	vfmach,vfmachr,vfmacw,vhfmach,vhfmachr,vhfmacw,vhmac,vlfmach,vlfmachr,vlfmacw,vlmac,vmac,vpfmachww,vxfmach,vxfmachr,vxfmacw,vxmac
SIMD difference of products instruction	X2	vfmsuh,vfmsuw,vhfmsuh,vhfmsuw,vhmsu,vlfmsuh,vlfmsuw,vlmsu,vmsu,vxfmsuh,vxfmsuw,vxmsu

It should be noted that "Operation units" in the above tables refer to operation units used in the respective instructions. More specifically, "A" denotes ALU instruction, "B" branch instruction, "C" conversion instruction, "DIV" divide instruction, "DBGM" debug instruction, "M" memory access instruction, "S1" and "S2" shift instructions, and "X1" and "X2" multiply instructions.

Fig.20 is a diagram showing the format of the instructions executed by the processor 1.

The following describes what acronyms stand for in the

diagrams: "P" is predicate (execution condition: one of the eight condition flags C0~C7 is specified); "OP" is operation code field; "R" is register field; "I" is immediate field; and "D" is displacement field.

Figs.21~36 are diagrams explaining outlined functionality of the instructions executed by the processor 1. More specifically, Fig.21 explains an instruction belonging to the category "ALUadd (addion) system)"; Fig.22 explains an instruction belonging to the category "ALUsub (subtraction) system)"; Fig.23 explains an instruction belonging to the category "ALUlogic (logical operation) system and the like"; Fig.24 explains an instruction belonging to the category "CMP (comparison operation) system"; Fig.25 explains an instruction belonging to the category "mul (multiplication) system"; Fig.26 explains an instruction belonging to the category "mac (sum of products operation) system"; Fig.27 explains an instruction belonging to the category "msu (difference of products) system"; Fig.28 explains an instruction belonging to the category "MEMld (load from memory) system"; Fig.29 explains an instruction belonging to the category "MEMstore (store in memory) system"; Fig.30 explains an instruction belonging to the category "BRA (branch) system"; Fig.31 explains an instruction belonging to the category "BSasl (arithmetic barrel shift) system and the like"; Fig.32 explains an instruction belonging to the category "BSasl (logical barrel shift) system and the like"; Fig.33 explains an instruction belonging to the category "CNVvaln (arithmetic conversion) system"; Fig.34 explains an instruction belonging to the category "CNV (general conversion) system"; Fig.35 explains an instruction belonging to the category "SATvlpk (saturation processing) system"; and Fig.36 explains an instruction belonging to the category "ETC (et cetera) system".

Figs.20 is a diagram showing the format of the instructions executed by the processor 1.

The following describes what acronyms stand for in the

diagrams: "P" is predicate (execution condition: one of the eight condition flags C0~C7 is specified); "OP" is operation code field; "R" is register field; "I" is immediate field; and "D" is displacement field.

Figs.21~36 are diagrams explaining outlined functionality of the instructions executed by the processor 1. More specifically, Fig.21 explains an instruction belonging to the category "ALUadd (addion) system)"; Fig.22 explains an instruction belonging to the category "ALUsub (subtraction) system)"; Fig.23 explains an instruction belonging to the category "ALUlogic (logical operation) system and the like"; Fig.24 explains an instruction belonging to the category "CMP (comparison operation) system"; Fig.25 explains an instruction belonging to the category "mul (multiplication) system"; Fig.26 explains an instruction belonging to the category "mac (sum of products operation) system"; Fig.27 explains an instruction belonging to the category "msu (difference of products) system"; Fig.28 explains an instruction belonging to the category "MEMld (load from memory) system"; Fig.29 explains an instruction belonging to the category "MEMstore (store in memory) system"; Fig.30 explains an instruction belonging to the category "BRA (branch) system"; Fig.31 explains an instruction belonging to the category "BSasl (arithmetic barrel shift) system and the like"; Fig.32 explains an instruction belonging to the category "BSasl (logical barrel shift) system and the like"; Fig.33 explains an instruction belonging to the category "CNVvaln (arithmetic conversion) system"; Fig.34 explains an instruction belonging to the category "CNV (general conversion) system"; Fig.35 explains an instruction belonging to the category "SATvlpk (saturation processing) system"; and Fig.36 explains an instruction belonging to the category "ETC (et cetera) system".

The following describes the meaning of each column in these diagrams: "SIMD" indicates the type of an instruction (distinction between SISD (SINGLE) and SIMD); "Size" indicates the size of

individual operand to be an operation target; "Instruction" indicates the operation code of an operation; "Operand" indicates the operands of an instruction; "CFR" indicates a change in the condition flag register; "PSR" indicates a change in the processor status register; "Typical behavior" indicates the overview of a behavior; "Operation unit" indicates a operation unit to be used; and "3116" indicates the size of an instruction.

The behaviors of the processor 1 concerning main instructions used in concrete examples that will be described later are explained below.

andn Rc,Ra,Rb

Carry out the inverted logical AND between Ra and Rb and store it in Rc.

asl Rb,Ra,I5

Execute an arithmetic shift left to Ra by only the number of bits in the immediate value (I5).

and Rb,Ra,I8

Carry out the logical AND between Ra and the value (I8) and store it in Rb.

bseq0 Rb,Ra

Count consecutive 0s from the MBS of Ra and store it in Rb.

bseq1 Rb,Ra

Count consecutive 1s from the MBS of Ra and store it in Rb.

bseq Rb,Ra

Count consecutive sign bits from 1 bit below the MSB of Ra and store it in Rb. When Ra is 0, output 0.

bcnt1 Rb,Ra

Count the number of 1s of Ra and store it Rb.

extr Rc,Ra,Rb

Designate the position of a bit by Rb, extract a part of contents of Ra, sign extend and store it in Rc.

extru Rc,Ra,Rb

Designate the position of a bit by Rb, extract a part of contents of Ra and store it in Rc without a sign extension.

fmulhh Mm,Rc,Ra,Rb

Treat Ra, Rb, and Rc as 16-bit values and treat Mm (an accumulator for multiplication) as a 32-bit value. Multiply Ra and Rb by a fixed point. Store the result in Mm and Rc. When the result cannot be represented by signed 32 bits, saturate it.

Fmulhw Mm,Rc,Ra,Rb

Treat Ra and Rb as 16-bit values and treat Mm and Rc as 32-bit values. Multiply Ra and Rb by a fixed point. Store the result in Mm and Rc. When the result cannot be represented by signaled 32 bits, saturate it.

mul Mm,Rc,Ra,Rb

Multiply Ra and Rb by an integer. Store the result in Mm and Rc.

mac Mm,Rc,Ra,Rb,Mn

Multiply Ra and Rb by an integer and add it to Mn. Store the result in Mm and Rc.

mov Rb,Ra

Transfer Ra to Rb.

or Rc,Ra,Rb

Carry out the logical OR between Ra and Rb and store it in Rc.

rde C0:C1,Rb,(Ra)

Let Ra be an external register number and read the value of the external register into Rb. Output a success and a failure of reading to C0 and C1 (condition flags), respectively. In the case of a failure, an exception of an expansion register error arises.

wte C0:C1,(Ra),Rb

Let Ra be an exernal register number and write the value of Rb into the external register. Output a success and a failure of writing to C0 and C1, respectively. In the case of a failure, an exception of an expansion register error arises.

vaddh Rc,Ra,Rb

Treat each register in half-word vector format. Add Ra and Rb (SIMD straight).

(A compiler)

Next, a compiler, according to the present embodiment, whose target is the above-described processor 1, is explained.

Fig. 37 is a function block diagram showing the configuration of a compiler 100 according to the present embodiment. This compiler 100 is a cross compiler that translates a source program 101 described in a high-level language such as C/C++ language into a machine language program 105 whose target processor is the above-described processor 1, is realized by a program executed on a computer such as a personal computer, and is largely divided into and configured with a parser unit 110, an intermediate code conversion unit 120, an optimization unit 130 and a code generation unit 140.

It should be noted that header files (an operator definition file 102 and a built-in function definition file 103) that efficiently generates the special-purpose instructions specific to the above-mentioned processor 1 is ready in the present compiler 100. A user can acquire the machine language program 105 specialized (optimized) for the processor 1 by including these header files in the source program 101.

The operator definition file 102 is, as shown in list examples in Fig. 38~Fig. 68, a header file that defines classes defining operators that targets fixed points and SIMD-type data. In the header file, Fig. 38~Fig. 40 are lists of a section where an operator whose target is mainly data of 16-bit fixed point of Mode 0 (_1 system) is defined; Fig. 41and Fig. 42 are lists of a section where an operator whose target is mainly data of 32-bit fixed point of Mode 0 (_1 system) is defined; Fig. 43~Fig. 45 are lists of a section where an operator

whose target is mainly data of 16-bit fixed point of Mode 1 (_2 system) is defined; Fig. 45~Fig. 47 are lists of a section where an operator whose target is mainly data of 32-bit fixed point of Mode 1 (_2 system); and Fig. 48~Fig. 68 are lists of a section where the other functions are defined.

The built-in function definition file 103 is, as shown in list examples in Fig. 69~Fig. 72, a header file that defines functions that execute various operations to replace the functions with machine language instructions specific to the processor 1. In the header file, Fig. 69~Fig. 71 are lists of a section where a function to replace the function with one machine language instruction is defined; Fig. 72 is a list of a section where a function to replace the functions with two or more machine language instructions (machine language instruction sequences) are defined.

It should be noted that asm(...){...}(...) in these definition files 102 and 103 is a built-in assembler instruction called an optimization asm, and is processed as follows. In other words, the description format of an optimization asm sentence is

asm(<<a list of load expressions >>){
<<optimization control information>>
<<instruction designation unit>>
}(<<a list of store expressions>>);.

Here, "the list of load expression" is a section where the load expressions are described; "the load expression" is an expression to store variables in C language and results of expressions such as four operations; it is described like "a register designation identifier = an assignment expression"; and it means that the value indicated on the right side is transferred to the identifier indicated in the left side. "The list of store expressions" is a section where the store expressions are described; "the store expression" is described like "a monomial = a register designation identifier"; and it means to assign a value on the left side represented by the monomial to a

value of the register represented by the register designation identifier.

The parser unit 110 is a front processing unit that extracts a reserved word (a keyword) and the like; carries out a lexical analysis of the source program 101 (that contains the header file to be included) that is a target of the compilation; and has a fixed point mode switch unit 111 that supports a switch of the mode on fixed points in addition to an analysis function that an ordinary compiler has. When the fixed point mode switch unit 111 detects in the source program 101 a pragma direction ("#pragma _save_fxpmode func", for example) that saves and restores the fixed point mode, it generates a machine language instruction that saves and restores a bit FXP of PSR31 of the processor 1. This realizes a programming in which the operations in both the Mode 0 and the Mode 1 of the fixed point mix.

It should be noted that a "pragma (or a pragma direction)" is a direction to the compiler 100 that the user can designate (place) arbitrarily in the source program 101 and a character sequence starting with "#pragma".

The intermediate code conversion unit 120 is a processing unit that converts each statement in the source program 101 passed from the parser unit 110 to intermediate codes and is made up of an intermediate code generation unit 121 and a machine language instruction substitution unit 122. The intermediate code generation unit 121 converts each statement in the source program 101 based on a predetermined rules. Here, an intermediate code is typically a code represented in a format of function invocation (a code indicating "+(int a, int b)"; indicating "add an integer a to an integer b", for example). But the intermediate code contains not only the code in the format of the function invocation but also machine language instructions of the processor 1.

The machine language instruction substitution unit 122

converts, out of the intermediate codes generated by the intermediate code generation unit 121, the intermediate codes in the format of the function invocation into the corresponding machine language instructions (or the machine language instruction sequence) referring to the operator definition file 102 and the built-in function definition file 103, and the intermediate codes that match the operators (including types of target data of the operations) defined by these definition files or the built-in functions into the corresponding machine language instructions (or the machine language instruction sequence) following a substitution table 122a, which the machine language instruction substitution unit 122 has inside of itself, or assembler instructions defined by these definition files, and outputs the converted machine language instructions to the optimization unit 130. This enables the optimization unit 130 to execute various optimizations to these intermediate codes because they are passed to the optimization unit 130 not in the format of the built-in functions but in the format of the machine language instructions.

By the way, the substitution table 122a is a table that stores the machine language instructions (or the machine language instruction sequence) corresponding to operations by operators reserved in advance and functions. Additionally, the machine language instruction substitution unit 122 outputs, out of the intermediate codes passed from the intermediate code generation unit 121, the machine language instructions without being processed to the optimization unit 130.

The optimization unit 130 is a processing unit that executes one of the below-mentioned three types of optimization selected by the user to the machine language instructions out of the intermediate codes outputted from the intermediate code conversion unit 120 by executing processing such as combining instructions, removing redundancy, sorting instructions and

allocating registers:(1) optimization in which increase of the execution speed has a higher priority; (2) optimization in which reduction of the code size has a higher priority; and (3) optimization of both the increase of the execution speed and the reduction of the code size. The optimization unit 130 has a processing unit (an argument optimization unit 131, a type conversion optimization unit 132 and a latency optimization unit 133) that performs the unique optimization to the present compiler 100 in addition to common optimization (such as "loop unrolling", "if conversion" and "generation of pair memory access instruction").

The argument optimization unit 131 is a processing unit that generates appropriate instructions or sequences (algorithm) according to the arguments of built-in functions (e.g. extr, extru). For example, when all the arguments are constants, the argument optimization unit 131 generates machine language instructions whose operands are the constant values acquired by holding in the constants; when a part of the arguments are constants, machine language instructions whose operands are immediate values; when all the arguments are variables, a sequence of instructions whose operands are registers.

The type conversion optimization unit 132 is a processing unit that makes operations between different types more efficient based on a certain notation in the source program 101. For example, when it is desirable that the multiplication result of a 16-bit data and another 16-bit data be kept as a 32-bit data, the type conversion optimization unit 132 generates one machine language instruction ("fmulhw" or the like) that executes the multiplication with such a type of conversion if there is the certain notation in the source program 101.

The latency optimization unit 133 aligns the machine language instructions based on a directive on latency (designation of the number of cycles) in an assembler instruction incorporated in

the source program 101 so that a specific section or a specific action takes execution time only the designated number of cycles. This makes it unnecessary for a programmer to do the conventional work that he inserts the required number of "nop" instructions and makes it possible to perform optimization by inserting other machine language instruction than the "nop" instruction.

By the way, "loop unrolling" is optimization that improves the possibility of parallel execution of a loop by expanding iteration (repetition) of the loop and generating a pair memory access instruction (ldp/stp/ldhp/sthp and the like) in order to execute a plurality of iterations at the same time. Additionally, an "if conversion" is optimization that removes a branch construction by generating an instruction for an execution mechanism with a condition (the instruction that is executed only when the condition (the predicate) included in the instruction matches the state of the processor 1 (the condition flag)). Moreover, "generation of a pair memory access instruction" is optimization that generates the pair memory access instruction (ldp/stp/ldhp/sthp and the like) with a pair register (two successive registers) as the target.

Furthermore, the optimization unit 130 outputs, out of intermediate codes of the function call format, the intermediate codes that cannot be expanded without being processed to the code generation unit 140 because it is impossible to perform the optimization processing at the above-described machine language instruction level.

The code generation unit 140 generates the machine language program 105, replacing all the intermediate codes (including codes of the function call format and optimized machine language instructions) outputted from the optimization unit 130, referring to a translation table and the like held internally.

Next, the characteristic behaviors of the compiler 100 configured as described above are explained indicating specific

examples.

Fig. 73 is a flowchart showing the behaviors of the machine language instruction substitution unit 122. The machine language instruction substitution unit 122 repeats the following process: (1) to judge whether the codes of the function call format out of the intermediate codes generated by the intermediate cord generation unit 121 match the operators (including the data type of an operation target) defined by the operator definition file 102 and the functions defined by the built-in function definition file 103 (Step S102) and, when they match (Yes at Step S101), (2) to substitute the operators and the functions with the machine language instructions (Step S102), following the assembler instructions defined by the substitution table 122a, which the machine language instruction substitution unit 122 has inside of itself, and these definition files 102 and 103 (Steps S100~S103).

To be more specific, implicit rules and the like of the type conversion among the different types are stipulated by the definitions of the operator definition file 102 (by the definitions of a constructor); the following four types of fixed points are defined:

"FIX16_1"; Signed 16 bits with the decimal point between the 14th bit and the 15th bit (MSB),

"FIX16_2"; Signed 16 bits with the decimal point between the 13th bit and the 14th bit,

"FIX32_1"; Signed 32 bits with the decimal point between the 30th and the 31st bit (MSB), and

"FIX32_2"; Signed 32 bits with the decimal point between the 29th bit and the 30th bit. Therefore, the machine language instruction substitution unit 122, for example, substitutes a source program

```
FIX16_1 a, b, c;
c = a * b;
```

with a machine language instruction

fmulhh m0,Rc,Ra,Rb (a fixed point multiplication operation

instruction).

Hereby, the user can declare that the four types, FIX16_1, FIX16_2, FIX32_1 and FIX32_2 are similar to the standard types of an ordinary compiler and use them. Then, the generated machine language instruction including the neighboring codes becomes an target of optimization such as combining instructions, removing redundancy, sorting instructions and allocating registers in the optimization unit 130, and can be optimized.

Similarly, the implicit rules and the like of the type conversion among the different types are stipulated by the definitions of the operator definition file 102 (by the definitions of a constructor); the following four types of SIMD instructions are defined:

"VINT8x4"; 8-bit integer data in 4 parallel,

"VINT16x2"; 16-bit integer data in 2 parallel,

"VFIX161x2"; 16-bit fixed point data of mode 0 (_1 system) in 2 parallel, and

"VFIX162x2"; 16-bit fixed point data of mode 1 (_2 system) in 2 parallel. Therefore, the machine language instruction substitution unit 122, for example, substitutes a source program

VINT16x2 a, b, c;

c = a + b;

with a machine language instruction

vaddh Rc,Ra,Rb (SIMD addition instruction).

Hereby, the user can declares that the four types, "VINT8x2", "VINT16x2", "VFIX161x2" and "VFIX162x2 are similar to the standard types of an ordinary compiler and use them. Then, the generated machine language instruction including the neighboring codes becomes an target of optimization such as combining instructions, removing redundancy, sorting instructions and allocating registers in the optimization unit 130, and can be optimized.

Additionally, in the built-in function definition file 103, a

function that can use advanced instructions the processor 1 executes ("_abs(a)" and the like, for example) and its corresponding advanced instruction (one machine language instruction "abs Rb, Ra" and the like, for example) are defined. Therefore, the machine language instruction substitution unit 122, for example, substitutes a source program

b = _abs(a);

with a machine language instruction

abs Rb,Ra.

Hereby, the user can realize a complex processing without creating one by C++language and an assembler instruction, but with only calling a built-in function prepared in advance. Then, the generated machine language instruction including the neighboring codes becomes an target of optimization such as combining instructions, removing redundancy, sorting instructions and allocating registers in the optimization unit 130, and can be optimized.

Similarly, in the built-in function definition file 103, a function that can use advanced instructions the processor 1 executes ("_div(a, b)" and the like, for example) and its corresponding advanced instruction (one machine language instruction sequence "extw, aslp, div" and the like, for example) are defined. Therefore, the machine language instruction substitution unit 122, for example, substitutes a source program

c = _div(a, b);

with a machine language instruction sequence

```
extw    Mn,Rc,Ra
aslp    Mn,Rc,Mn,Rc,15
div     MHm,Rc,MHn,Rc,Rb.
```

Hereby, the user can realize a complex processing without creating one by C++language and an assembler instruction, but with only calling a built-in function prepared in advance. Then, the

generated machine language instruction s equence including the neighboring codes becomes an target of optimization such as combining instructions, removing redundancy, sorting instructions and allocating registers in the optimization unit 130, and can be optimized.

It should be noted that out of the built-in functions listed in the built-in function definition file 103, representative examples (especially effective for media processing) of (1) functions that are converted into one machine language instruction, (2) functions that are converted into two or more machine language instructions (a machine language instruction sequence) and (3) functions that can designate resources that are not targets of the register allocation (such as an accumulator) are as follows:

(1) The built-in functions that are converted into one machine language instruction

"_bseq1(x)":

This is the function that detects how many bit 0s succeed from the MSB of the input. The formats are as follows:

int _bseq1(FIX16_1 val) // count 1
int _bseq1(FIX16_2 val) // count 1
int _bseq1(FIX32_1 val) // count 1
int _bseq1(FIX32_2 val) // count 1

These functions return the value of the number of successive 0s (the number of bits) in "val" to be counted. The machine language instructions corresponding to these functions are defined in the built-in function definition file 103.

"_bseq0(x)":

This is the function that detects how many bit 0s succeed from the MSB of the input. The formats are as follows:

int _bseq0(FIX16_1 val) // count 0
int _bseq0(FIX16_2 val) // count 0
int _bseq0(FIX32_1 val) // count 0

int _bseq0(FIX32_2 val) // count 0

These functions return the value of the number of successive 0s (the number of bits) in "val" to be counted. The machine language instructions corresponding to these functions are defined in the built-in function definition file 103.

"_bseq1(x)":

This is the function that detects how many bit 1s succeed from the MSB of the input. The formats are as follows:

int _bseq1(FIX16_1 val) // count 1
int _bseq1(FIX16_2 val) // count 1
int _bseq1(FIX32_1 val) // count 1
int _bseq1(FIX32_2 val) // count 1

These functions return the value of the number of successive 1s (the number of bits) in "val" to be counted. The machine language instructions corresponding to these functions are defined in the built-in function definition file 103.

"_bseq(x)":

This is the function that detects how many bits of the same value as the MSB succeed from the next bit to the MSB of the input. The formats are as follows:

int _bseq(FIX16_1 val)
int _bseq(FIX16_2 val)
int _bseq(FIX32_1 val)
int _bseq(FIX32_2 val)

These functions return the number of the normalized bits of "val". The machine language instructions corresponding to these functions are defined in the built-in function definition file 103.

"_bcnt1(x):

This is the function that detects how many bit 1s are included in the all bits of the input. The formats are as follows:

int _bcnt1(FIX16_1 val)
int _bcnt1(FIX16_2 val)

int _bcnt1(FIX32_1 val)

int _bcnt1(FIX32_2 val)

These functions return the value of the number of 1s in "val" to be counted. The machine language instructions corresponding to these functions are defined in the built-in function definition file 103.

"_extr(a,i1,i2)":

This is the function that extracts and sign-expands the predetermined bit positions of the input. The formats are as follows:

int _extr (FIX16_1 val1, int val2, int val3)

int _extr (FIX16_2 val1, int val2, int val3)

int _extr (FIX32_1 val1, int val2, int val3)

int _extr (FIX32_2 val1, int val2, int val3)

These functions return the result that the bit field of val1 indicated from the bit position va12 to the bit position va13 is extracted and sign-expanded. The machine language instructions corresponding to these functions are defined in the built-in function definition file 103.

"_extru(a,i1,i2)":

This is the function that extracts and zero-expands the predetermined bit positions of the input. The formats are as follows:

unsigned int _extru (FIX16_1 val, int val2, int val3)

unsigned int _extru (FIX16_2 val, int val2, int val3)

unsigned int _extru (FIX32_1 val, int val2, int val3)

unsigned int _extru (FIX32_2 val, int val2, int val3)

These functions return the result that the bit field of val1 indicated from the bit position va12 to the bit position va13 is extracted and zero-expanded. The machine language instructions corresponding to these functions are defined in the built-in function definition file 103.

(2) The built-in functions that are converted into two or more machine language instructions (a machine language instruction sequence)

"_modulo_add()":

This is the function that performs an address update of the modulo addressing. The format is as follows:

_modulo_add(void *addr, int imm, int mask, size_t size, void *base)

Here, the meaning of each argument is as follows:

addr: the address before the update or the lower address (the modulo part)

imm: the value of addition (the number of data)

mask: the width of mask (the width of modulo)

size: the size of data (exponentiation of two)

base: the base address (the head address of the array)

This function returns the result that only the addition value imm is added from the address addr by the modulo addressing.

The machine language instructions corresponding to this function is defined in the built-in function definition file 103. In other words, this function uses the instruction (addmsk) that substitutes the predetermined bit field of the first input with the predetermined bit field of the second input to calculate the modulo addressing. A usage example is as follows:

```
int  array[MODULO];
p = array;
for (i = 0; i < 100; i++) {
*q++ = *p;
p = (int *)_modulo_add(p, 1, N, sizeof(int), array);
}
```

Here, the variable MODULO is the exponentiation of two (2^N). In this usage example, 100 elements of the array are placed with the alignment of the MODULO*SIZE bite.

"_brev_add()":

This is the function that performs an address update of the bit reverse addressing. The format is as follows:

_brev_add(void *addr, int cnt, int imm, int mask, size_t size, void *base)

Here, the meaning of each argument is as follows:

addr: the address before the update

cnt: bit reverse counter

mm: the value of addition (the number of data)

mask: the width of mask (the width of reverse)

size: the size of data (exponentiation of two)

base: the base address (the head address of the array)

This function returns the result that only the addition value mm is added from the address addr that corresponds to the bit reverse counter cnt by the bit reverse addressing. .

The machine language instructions corresponding to this function is defined in the built-in function definition file 103. In other words, this function uses the instruction (mskbrvh) that executes a position reverse bit by bit to the predetermined bit field of the first input to calculate the bit reverse addressing. A usage example is as follows:

```
int  array[BREV];
p = array;
for (i = 0; i < 100; i++) {
*q++ = *p;
p = (int *)_brev_add(p, i, 1, N, sizeof(int), array);
}
```

Here, the variable BREV is the exponentiation of two (2^N). In this usage example, 100 elements of the array are placed with the alignment of the BREV*SIZE bite.

(3) Functions that can designate resources that are not targets of the register allocation (such as an accumulator)

In the built-in function definition file 103, in addition to a

general-purpose register that is a target resource of the register allocation in the optimization, the built-in functions (multiplication: "_mul" and product-sum operation: "_mac") that (i) are operations (multiplication and product-sum operation) to update also the accumulator that is not a target of register allocation (that is implicit resources) and (ii) can designate a temporary variant with an accumulator as a reference type are prepared. The concrete formats are as follows respectively:

_mul(long &mh, long &ml, FIX16_1 &c, FIX16_1 a, FIX16_1 b);

This function multiplies a variant a and a variant b together, sets up the higher 32 bits of 64-bit data, which is the result, to the higher accumulator for multiplication MH, the lower 32 bits of the 64-bit data to the lower accumulator for multiplication ML, and further, 32-bit data that combine the lower 16 bits of the accumulator MH and the higher 16 bits of the accumulator ML to a variant c.

_mac(long &mh, long &ml, FIX16_1 &c, FIX16_1 a, FIX16_1 b);

This function adds the 64-bit data that combine the higher accumulator for multiplication MH and the lower accumulator for multiplication ML to the result gained by multiplying the variant a and the variant b, and sets up the higher 32 bits of 64-bit data, which is the result, to the higher accumulator for multiplication MH, the lower 32 bits of the 64-bit data to the lower accumulator for multiplication ML, and further, 32-bit data that combine the lower 16 bits of the accumulator MH and the higher 16 bits of the accumulator ML to a variant c.

A usage example is as follows:

The machine language instruction substitution unit 122 substitutes the following source program

long mh,ml;
_mul(mh,ml,dummy,a,b);
_mac(mh,ml,e,c,d);

with the following machine language instruction

mul m0,Rx,Ra,Rb

mov r0,mh0

mov r1,mh1

mov mh0,r0

mov mh1,r1

mac m0,Re,Rc,Rd,m0

following the definitions in the built-in function definition file 103. It should be noted that, out of the above-mentioned machine language instruction sequence, the first to the third lines correspond to the function_mul and the fourth to the sixth lines correspond to the function_mac. The second to the fifth lines of the machine language instruction sequence like this are deleted by removing redundancy in the optimization unit 130 and the machine language instruction sequence is optimized to the following machine language instruction sequence

mul m0,Rx,Ra,Rb

mac m0,Re,Rc,Rd,m0.

As is described above, when the functions that can designate resources that are not targets of the register allocation (such as an accumulator) are used, it is highly probable that a set of the definition (storage of the value) and the usage (reference to the value) is deleted by the optimization within the compiler (the optimization unit 130) and therefore the built-in function like this is effective in terms of the optimization, too.

Next, behaviors by the optimization unit 130 out of the characteristic behaviors of the present compiler 100 are explained.

Fig. 74 is a flowchart showing the behaviors of the argument optimization unit 131 of the optimization unit 130. To generate an appropriate instructions or an appropriate sequence (algorithm) depending on the arguments, the argument optimization unit 131 generates:

(1) a machine language instruction using the constant values acquired by folding in the constants set out below as operands (Step S111) when all the arguments of the function are constants (left at Step S110);

(2) a machine language instruction of an immediate operand (Step 112) when a part of the arguments are constants (center at Step S110); and

(3) a sequence of machine language instructions of a register operand (Step 113) when all the arguments are variables (right at Step S110).

For example, when all the arguments are constants like

d = _extru(0xfff, 7, 4);

a machine language instruction with the constant values acquired by folding in the constants like

movRd,0xf

is generated.

On the other hand, when a part of the arguments are constants like

d = _extru(a, 7, 4);

a machine language instruction of an immediate operand like

extruRd,Ra,7,4

is generated.

Further, when all the arguments are variables like

d = _extru(a, b, c);

a sequence of machine language instructions like

aslRe,Rb,8

andRf,Rc,0x1f

orRg,Re,Rf

extruRd,Ra,Rg

is generated.

As just described, from one built-in function, the same machine language instruction is not always generated fixedly but a

machine language instruction (or a sequence of machine language instructions) optimized depending on the nature of the argument is generated by the argument optimization unit 131.

Fig. 75 is a diagram showing an arithmetic tree to explain behaviors of the type conversion optimization unit 132 of the optimization unit 130. The type conversion optimization unit 132 generates a machine language instruction with the type conversion (such as fmulhw) to the operations of a certain notation in the source program to perform the efficient operations among different types.

In the ordinary C language, the type of the result of 16bit x 16bit is 16bit. An instruction of 16bit x 16bit --> 32bit exists, but two different machine language instructions are generated as described below. For example, to the description of

f32 = f16 * f16;

two instructions are generated:

```
fmulhh    // 16bit x 16bit -> 16bit
asl       // 16bit -> type conversion of 32bit
```

Consequently, when (FIX32)16bit * (FIX32)16bit is described in the source program, the type conversion optimization unit 132 ordinarily generates an arithmetic tree shown in Fig. 75A (a cord of the type conversion is generated) but one instruction (fmulhw) of 16bit x 16bit --> 32bit by converting this arithmetic tree into the arithmetic tree shown in Fig. 75B.

Fig. 76 is a diagram showing an example of a sample program to explain behaviors of the latency optimization unit 133. The latency optimization unit 133 executes scheduling a machine language instruction based on a directive concerning latency (designation of the number of cycles) in an assembler instruction (an optimized asm sentence) built in the source program 101 so that it takes execution time only the designated number of cycles in behaviors in a specific section or a specific behavior.

The user can set up the latency by two kinds of designation methods.

One method is to designate the latency among labels attached to the specific instructions like the designation in the program (LATENCY L1, L2, 2;) shown in Fig. 76A. In the example of Fig. 76A, the latency optimization unit 133 executes scheduling the allocation of the machine language instruction sequence so that only 2 cycles pass since the instruction wte is executed until the instruction rde is executed by the processor 1.

The other method is to designate the latencty on the instructions (rd, wt, rde, wte) that access the expansion register unit 80 until the instructions access the expansion register unit 80 the next time like the designation in the program (LATENCY (2) inside the instruction wte) shown in Fig. 76B. In the example of Fig. 76B, the latency optimization unit 133 executes scheduling the allocation of the machine language instruction sequence in order that only 2 cycles pass since the instruction wte is executed by the processor 1 and the expansion register unit 80 is accessed until the expansion register unit 80 is accessed again.

With the configuration of the latency like this, it is possible to execute the optimization (combining instructions, removing redundancy, sorting instructions and allocating registers) between the codes that have been in-line expanded and the codes that have not yet been in-line expanded and the latency between the designated instructions or accesses is secured. In other words, conventionally the user must insert explicitly a nop instruction but when he uses the compiler 100, all he must do is to designate a necessary latency for a necessary instruction or a necessary access.

Fig. 77 is a diagram explaining behaviors of the fixed point mode switch unit 111 of the parser unit 110.

When the fixed point mode switch unit 111 detects a pragma directive to save and return the fixed point mode ("#pragma

_save_fxpmode func", for example) in the source program 101, the fixed mode switch 111 generates a machine language instruction to save and return the bit FXP of PSR 31 of the processor 1.

It should be noted that as the specifications on the fixed point that are premises, a _1 system (FIX16_1, FIX32_1) type and a _2 system (FIX16_2, FIX32_2) type exist; the mode is switched by one bit (FXP) of PSR31 in hardware (the processor 1); and further a condition that only single system can be used within a function exists.

Consequently, as the method for switching and using these two systems on the program, it is made a rule to designate a pragma ("#pragma _save_fxpmode" the name of a function) as a function that may be called by other systems. Hereby, the fixed point mode switch unit 111 inserts the codes corresponding to the save and the return of the FIX-type mode into the head and the tail of the function. Additionally, a FIX-type declaration of each function is searched; by which FIX-type declaration the function is compiled is decided; and the code to set up the mode is inserted.

Fig. 77A shows an example of a function with the pragma directive. The comment written in the right side of Fig. 77A is the insertion processing by the fixed point mode switch unit 111 and its concrete processing is shown in Fig. 77B.

An applied example of the pragma directive like this is shown in Fig. 77C. For example, regarding four functions, f11, f21, f22 and f 23, when the function f11: _1 system calls the function f21: _2 system; the function f21: _2 system calls the function f22: _2 system; the function f22: _2 system calls the function f23: _2 system, since the only function that may be called by other modes is f21, it is possible to switch to a normal mode by executing a pragma designation only to this function.

As described above, using the compiler 100 according to the present embodiment, by cooperative processing between the

operator definition file 102, the built-in function definition file 103 and the machine language instruction substitution unit 122, the user can declare and use the fixed point types of mode 0 and mode 1 as ordinary types and generate effectively high-functional machine language instructions that the processor 1 executes by calling the built-in functions at the level of a high-level language.

Additionally, with optimization of the arguments of the built-in functions by the argument optimization unit 131, machine language instructions with effective operands are generated. Moreover, with optimization of the type conversion by the type conversion optimization unit 132, an operation with a type conversion is converted into one high-functional machine language instruction that the processor 1 executes. Further, with the scheduling of machine language instructions by the latency optimization unit 133, the user can secure latency in the access between the specific instructions or to the expansion register without inserting the nop instruction.

Up to this point, the compiler according to the present invention has been explained based on the embodiment, but the present invention is not limited by this embodiment.

For example, in the present embodiment, the types of the fixed point are 16 bits or 32 bits and the decimal point is placed at the MSB or its lower digit, but the present invention is not limited to the format like this and it is acceptable that the type whose fixed points are 8 bits or 64 bits and whose decimal point is placed at another digit is a target.

Additionally, it is also acceptable to offer a behavior verification technique using a class library as a development support tool for the user. In other words, ordinarily, as is shown in Fig. 78A, the test source and the definition files 102 and 103 are compiled using the cross compiler (the compiler 100) for the target machine (the processor 1) according to the present embodiment; the

behavior verification is performed by executing the obtained machine language program for the processor 1 with the special-purpose simulator. Instead of this, as is shown in Fig. 78B, it is acceptable that a class library (a definition file that associates the operator definition file 102 and the built-in function definition file 103 respectively with machine language instructions of a host machine not the processor 1) whose target is a host machine for development (a processor produced by Intel Corporation, for example) is prepared and compiled together with the test source, the definition files 102 and 103 by a native compiler (such as Visual C++ (R)), and the host machine executes the obtained machine language program as-is. Hereby, it is possible to execute a simulation in the familiar environment at high speed and perform the behavior verification.

Further, in the present embodiment, the operators and the built-in functions that are associated with the machine language instructions that are specific to the target processor are supplied as the header files (the definition files 102 and 103), but the compiler according to the present invention may be configured to incorporate the information of the definition files like these in the compiler itself. In other words, the compiler according to the present invention is an integral-type program into which the above-mentioned definition files 102 and 103 are incorporated and it is acceptable that the compiler is configured in order that the compiler that translates a source program into a machine language program, the program including operation definition information in which operation that corresponds to a machine language instruction specific to a target processor is defined, the compiler comprising: a parser step of analyzing the source program; an intermediate code conversion step of converting the analyzed source program into intermediate codes; an optimization step of optimizing the converted intermediate codes; and a code generation step of converting the

optimized intermediate codes into machine language instructions, wherein the intermediate code conversion step includes: a detection sub-step of detecting whether or not any of the intermediate codes refer to the operation defined in the operation definition information; and a substitution sub-step of substituting the intermediate code with a corresponding machine language instruction, when the intermediate code is detected, and in the optimization step, the intermediate codes are optimized, the intermediate codes including the machine language instruction substituted for the intermediate code in the substitution sub-step. Hereby, the user does not need to include the definition files in the source program.